Date __/__/____ Searching, Sorting. Saathi

HashMap, Sets, Hashing & Collision Resolution & Storage Management

Linear Search :-

The most simplest type of search algorithm is the linear search. Generally, we are interested in searching any data values in any of the data structures.

The basic idea of linear search is -

Traversing the array one by one and comparing each element by the key element and when it is found, return the index, etc.

Code :-

```
for(int i=0; i<size; i++){
    if(array[i] == target){
        return i;
    }
}
```

Binary Search :- The fastest searching Algorithm with Time Complexity O(log n). It follows divide & Conqueror rule but the only condition is that the searching array or list or anything must be sorted in any order either decreasing or increasing

finding → high, low value then finding → middle.

initially

low → 0 ; → index value.

mid → low + (high - low) / 2 ; To prevent out of bound error

high → (size) - 1 ; index value or data type overflow.

Page No.

Date ___ / ___ / _____

```
while(low<=high){  if(arr[mid] == key){
                       return mid ; // for index of element
                       return arr[mid]; // for searched element
                   }
                   if(arr[mid] > key){
                       high = mid-1;
                   }
                   else{
                       low = mid+1;
                   }
                }
```

example

key → 1

1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7

low = 0
high = 7
mid = 0 + (7-0)/2
= 3

mid

arr[mid] > key.
high = mid - 1

1	2	3
0	1	2

low = 0
high = 2
mid = 1

mid

arr[mid] > key
high = mid - 1

[1] → arr[mid] == key

low
high
mid

Saathi

Date __/__/____

Sorting →

(1) Bubble Sort (2) Insertion Sort (3) Selection Sort
(4) Quick Sort (5) Merge Sort (6) Heap Sort
(7) Shell Sort (8) Bucket Sort (9) Radix Sort
(10) Counting Sort (11) Comparison Sort (12) External Sort
(13) Integer Sort (14) Stable Sort.

(1) **Bubble Sort** → Simplest technique to sort any array of given elements.

The number of passes is equal to the number of element in array.

We go left to right compare each two adjacent elements.

example → { 6, 18, 13, 11, 4 }.

1. 6 & 18 no swap.
2. 18 & 13 swap → 13, 18.
3. 18 & 11 swap → 11, 18
4. 18 & 4 swap → 4, 18.

{ 6 13 11 4 18 },

1. 6 & 13 no swap.
2. 13 & 11 swap → 11, 13
3. 13 & 4 swap → 4, 13.
4. 13 & 18 no swap.

{ 6, 11, 4, 13, 18 }.

1. 6 & 11 no swap.
2. 11 & 4 swap → 4, 11
3. 11 & 13 no swap.
4. So, ...

→ { 6, 4, 11, 13, 18 }.

1. 6 & 4 → swap 4, 6

{ 4, 6, 11, 13, 18 }.

Page No.

Date ___/___/_____

Saathi

optimised Code.

```
void bubbleSort (int arr[], int n){
    int passes = n;
    for( int i=0; i< passes; i++){
        bool flag = false;
        for( int j=0; j< n-1; j++ ){
            if (arr[j] > arr[j+1]){
                Swap (arr[j], arr[j+1]);
                flag = true;
            }
        }
        if (flag == false) {
            break;
        }
    }
}
```

Time Complexity → $O(n^2)$ after optimization. average case T.C can be improved.

Selection Sort → Idea is to place minimum element to its original index first in the array.

* No. of passes = no. of element in the array - 1
* left to right traversal to find the min element from that index to the last element of the array.

Date ___/___/_____

Saathi

example = { 6, 10, 13, 11, 4 }

(1) pass → 4, 10, 13, 11, 6 { swap → 6, 4 }
(2) pass → 4, 6, 13, 11, 10 { swap → 6, 10 }
(3) pass → 4, 6, 11, 13, 10 { swap → 11, 13 }
(4) pass → 4, 6, 11, 13, 10 { No swap }

Optimised Code

```
void SelectionSort(int arr[]){
    int size = sizeof(arr)/sizeof(arr[0]);
    size = size - 1;
    int n = size + 1;
    for(int i=0; i<=size; i++){
        int min_index = i;
        for(int j=i; j<n; j++){
            if(arr[j] < arr[min_index]){
                min_index = j;
            }
        }
    if(min_index != i) swap(arr[i], arr[min_index]);
    }
}
```

T.C. = $O(n^2)$

Date ___/___/_____

Insertion Sort :→ It is based on the idea of already sorted array you need to add another element into it such that the resultant array is also sorted.

(1) All elements in left are less than equal to the given element to be inserted.

(2) All the element to the right are greater than or equal to the given element to be inserted.

Algorithm →

(1) In this, we compare all the elements of that array one by one with the previous elements.

(2) If at any point of time, the previous element becomes smaller than the current element then we stop comparing and put the current element at that index.

(3) We stop because we know that the left part of the array has already become sorted in the previous passes.

(4) Thus the current element comes at the correct position in the array.

Date __/__/____

Saathi

Example → {6, 10, 13, 11, 4}

Compare → 6 & 10 {No Swap}
Compare → 10 & 13 {Swap (10, 13) then we check 13 with 6. {no Swap}}
Compare → 10 & 11 {Swap (10, 11) then 11 & 13 (Swap) then 11 & 6 {no Swap}}.

Now → {6, 11, 13, 10, 4}
Compare → 10 & 4 {Swap} then 4 & 13 (Swap) 4 & 11 (Swap) 4 & 6 (Swap)

{4, 6, 11, 13, 10}

Time Complexity $O(n^2)$

Code.

```
void insertionSort(int arr[], int n) {
    for (int i=1; i<n; i++) {
        int curr_e = arr[i];
        int j = i-1;
        while (j>=0 && arr[j] > curr_e) {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = curr_e;
    }
}
```

Saathi

Date __/__/____

Quick Sort → Fastest sorting algorithm with time complexity (nlog n) but in worst case it is O(n²)

Best Case O(n log n)
Worst Case O(n²)
Average Case O(n log n)

Space complexity →
O(log(n)) in the average case.
O(n) in worst case due unbalanced partitioning.

It is Based on Divide & Conqueror technique.

Algorithm

(1) Choose a pivot then place it on its correct location. such that left side of the pivot is lesser & right side is greater
(2) Then create partition according to pivot
(3) Rest of the condition is for Recursion.

Example :-

```
if (arr[j] < key/pivot) {
    i++;
    swap (arr[i], arr[j]);
    j++;
}
else {
    j++;
```

Page No.

Saathi

Date __/__/__

Size → 10 (1) 42, 84, 75, 20, 60, 10, 90, 50, 5, 30

key/pivot/arr[i] → 42. arr[j] = 84.

j++.

(2) 42, 84, 75, 20, 60, 10, 90, 50, 5, 30

j++.

(3) 42, 84, 75, 20, 80, 10, 90, 50, 5, 30.

key > arr[j]

i++;

Swap(arr[i], arr[j]);

j++;

(4) 42, 20, 75, 84, 60, 10, 90, 50, 5, 30.

(5) 42, 20, 75, 84, 60, 10, 90, 50, 5, 30

j++

(6) 42, 20, 75, 84, 60, 10, 90, 50, 5, 30.

i++;

Swap(a[i], a[j]);

j++;

42 20 10 84, 60, 75, 90, 50, 5, 30.

j++

(7) 42 20 10 84 60 75, 90, 50, 5, 30

(8) 42 20 10 84 60 75, 90, 50, 5, 30.

key 42 > arr[j] 5.

i++

swap(i,j); j++;

Page No.

Date __/__/____

Saathi

(a) 42, 20, 10, 5, 60, 75, 90, 50, 84, 30
i = 0, j = 0

(b) 42 > 30.
i++;
Swap(arr[i], arr[j]);
j++;

42, 20, 10, 5, 30, 75, 90, 50, 84, 60

if (j == Size - 1)
Swap (a[i] & key)

30, 20, 10, 5, 42, 75, 90, 50, 84, 60
values < pivot | pivot | values > pivot

after Recursive call

After Recursive call.

5, 10, 20 30, 42, 75, 90, 50, 84, 60.

Page No.

Date __/__/____

Saathi

{ Code method is a bit different }:

Code

```
void QuickSort ( int arr [], int s, int e) {
    if ( s >= e) {
        return ; }
    int p = partition ( arr , s, e) ;
    quickSort ( arr, s, p-1);
    quickSort ( arr, p+1, e) ;
}

int partition ( int arr[], int s, int e) {
    int pivot = arr[s];
    int cnt = 0 ;
    for ( int i = s+1 ; i <= e ; i++) {
        if ( arr [i] <= pivot) {
            cnt++;
        } }
    int pivotIndex = s + cnt ;
    swap ( arr [pivotIndex ], arr [s]) ;
    int i = s , j = e ;
    while ( i < pivotIndex && j > pivotIndex) {
        while ( arr[i] < pivot) {
            i++ ; }
        while ( arr[j] > pivot ) { j-- ; }
        if ( i < pivotIndex , && j > pivotIndex) {
            swap ( arr [i++] , arr [j--]) ;
        } }
```

Date ___/___/_____

Saathi

Merge Sort → Best Sorting Algorithm in terms of Time complexity as it provides :

$O(N \cdot \log N)$ in all the cases.

Best → $\Omega(N \cdot \log n)$

Average → $\Theta(N \cdot \log n)$

Worst → $O(N \cdot \log n)$

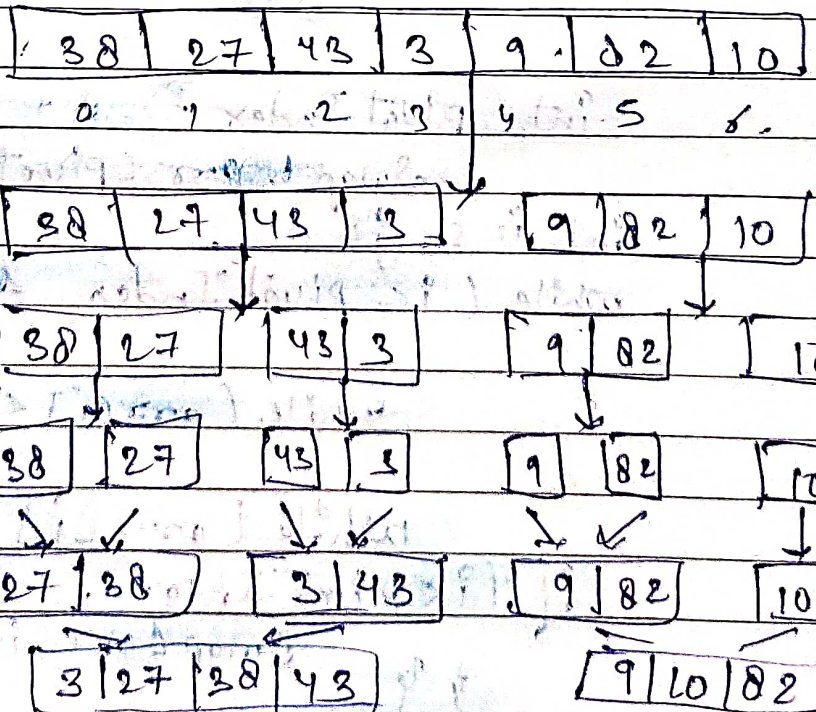
But only flaws of the algorithm is it take $O(n)$ space which leads to slower operation in last data sets.

It is based on Divide & Conqueror technique.

Algorithm :

1. Divide the array upto undividable part
2. Merge the array and perform the comparison while merging.

Example →



Date ___/___/_____

Saathi

3	9	10	27	38	43	82

Code

```
void MergeSort(int arr[], int s, int e){
    if(s>=e){
        return;}
    mergeSort(arr, s, s+(e-s)/2);
    mergeSort(arr, s+(e-s)/2+1, e);

    mergeSortArray(arr, s, e);
}

void MergeSortArray(int arr[], int s, int e){
    int mid = s+(e-s)/2;
    int len1 = mid-s+1;
    int len2 = e-mid;
    int *first = new int[len1];
    int *second = new int[len2];
    int mainArrayIndex = 0;
    for(int i=0; i<len1; i++){
        first[i] = arr[mainArrayIndex++];
    } mainArrayIndex = mid+1;
    for(int i=0; i<len2; i++){
        second[i] = arr[mainArrayIndex++];
    }
    int i=0, j=0;
    mainArrayIndex = s;
```

```
while (i < len1 && j < len2) {
    if (first[i] < second[j]){
        arr[mainArrayIndex++] = first[i];
        mainArrayIndex++;
        i++;
    } else {
        arr[mainArrayIndex++] = second[j++];
    }
while (i < len1) {
    arr[mainArrayIndex++] = first[i++];
    }
while (j < len2) {
    arr[mainArrayIndex++] = second[j++];
    }
}
```

Applications for Merge Sort →

1. Linked list Sort in O(nlogn)
2. Inversion Count Problem.
3. External Sorting

T.C = (nlogn)
S.C → O(n)

why we prefer.

Merge Sort in Linked List
Quick Sort in Arrays.

Date __/__/____

Saathi

HeapSort

For HeapSort we first have to understand heapify

Heapify ~~Algo~~ :→

In a CBT → $\left(\frac{n}{2}+1\right) \longrightarrow (n)$

are all leaf nodes we need not to process all these.

we have to process $\{1 \to n/2\}$ part.

process :→ putting nodes to there correct place.

Heapify → { for(i=n/2 ; i>0) {
placing nodes to their right place }; }

```
void heapify(int arr[], int size, int index){
    int largest = index;
    int left = 2*index+1;
    int right = 2*index+2;

    if(left<=size && arr[left] > arr[largest]){
        largest = left;
    }
    if(right<=size && arr[right] > arr[largest]){
        largest = right;
    }
    if(smallest != index){
        swap(arr[index], arr[largest]);
        heapify(arr, size, largest);
    }
```

0 based
left = 2*i+1;
right = 2*i+2;

1 based indexing

1 based indexing (<=)

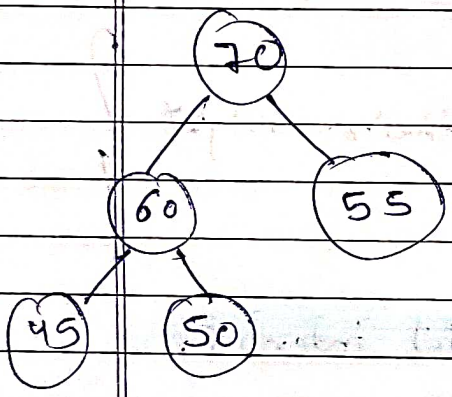
0 based (<)

Date ___/___/_____

```
int main(){
    // as we know we need not to
    // calculate the leaf node then we use
    for(int i= n/2 ; i>=0 ; i--){
        heapify(arr, size, i);
    }
}
```

1 → based indexing

HeapSort
O(nlogn)



(1) This sorting is based on heapify algorithm completely.

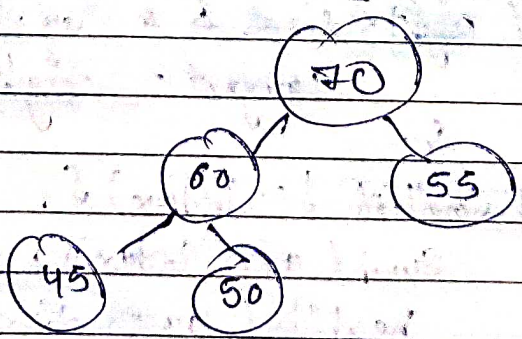
(2) Swap root element with right most leaf if not available the left, but from right side.

(3) reduce size as 1 element is sorted.

(4) Call heapify function to rearrange.

(5) Cont. until the array is sorted.

Visualize

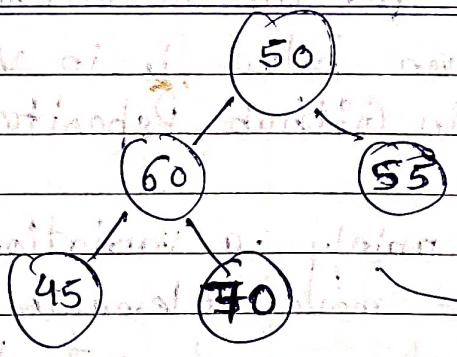


Swap(50, 70)

Page No.

for more clarity check V.S. code (Saathi)

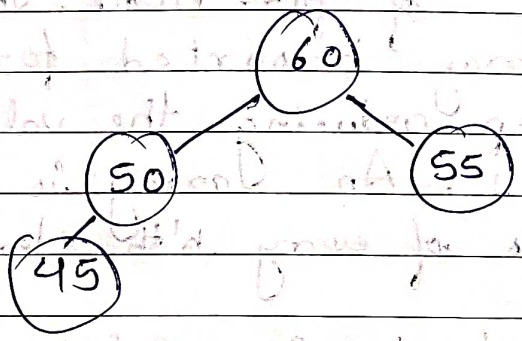
Date __/__/____



Remove last
i.e. → size--

50 60 55 45 | 70
size--

(2) Heapify 50. then.



60, 50, 55, 45 | 70

Cont.

(3) Swapping → 45, 50, 55 | 60, 70

(4) Heapify → 55, 50, 45 | 60, 70.

(5) Swapping → 45, 50 | 55 60 70.

(6) Heapify → 50, 45 | 55 60 70.

(7) Swapping → 45 | 50 55 60 70.

(8) Heapify → | 45 | 50 | 55 | 60 | 70 |

```
void heapsort( int arr[], int size){
    // Build heap.
    int i = size;
    while (i > 1) {
        swap(arr[i], arr[1]);
        i--;
        heapify (arr, size, 1);
    }
}
```

Page No.

Saathi

Date __/__/____ Code for all the Sorting given below is in Vscode or in Github Repository.

Shell Sort → It is mainly a variation of Insertion Sort, we move elements only one position ahead. when an element has to be moved far ahead, many movements are involved. The idea of shellsort is to allow the exchange of far items. In shell sort, we make the array h-sorted for a large value of h. we keep reducing the value of h until it become 1. An array is said to be h-sorted if all sublists of every h'th element are sorted.

$O(n^2)$ $\Omega(n \log n)$ $\Theta(n \lg n) \sim \Theta(n^{1.25})$

Bucket Sort → It is a sorting technique that involves dividing elements into various groups, or buckets. These buckets are formed by uniformly distributing the elements. Once the elements are dividing into buckets, they can by sorted using another sorting algorith.

It create n empty buckets and do the followin for every array element arr[i].

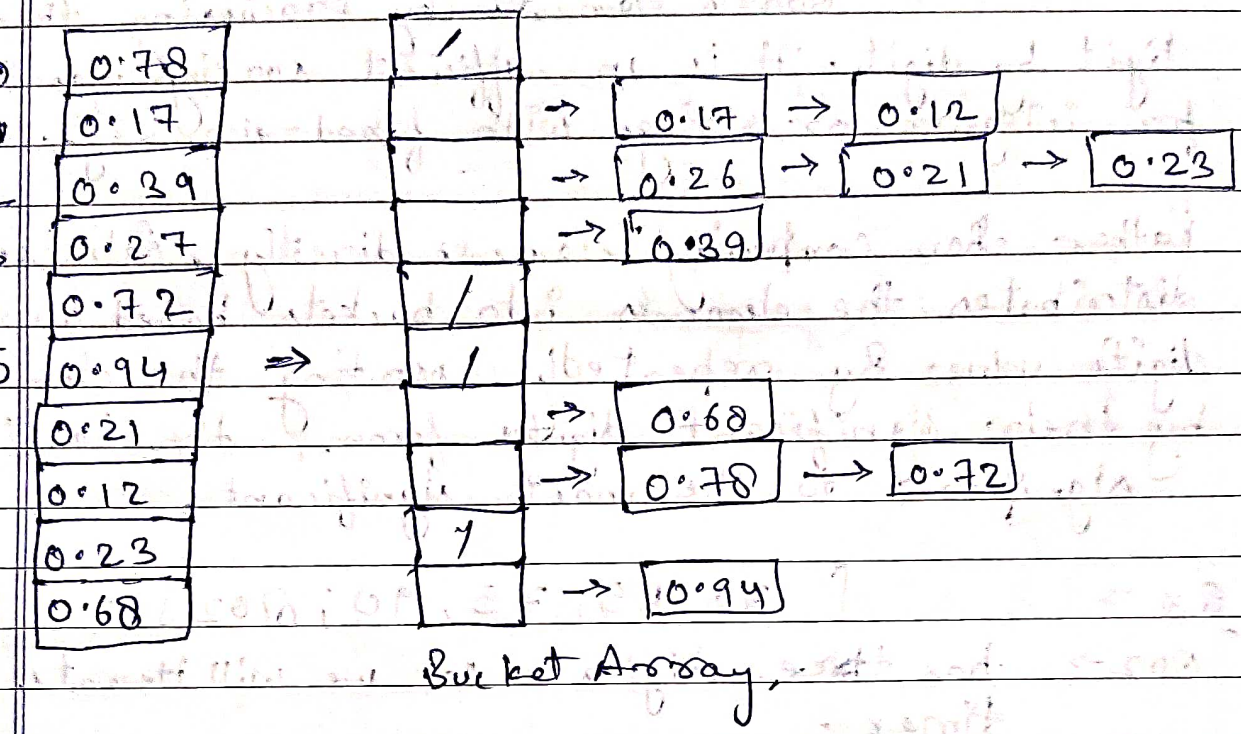
- Insert arr[i] into bucket[n * array[i]]
- Sort individual bucket using Insertion Sort
- Concatenate all Sorted buckets.

Page No.

Date __/__/____

Saathi

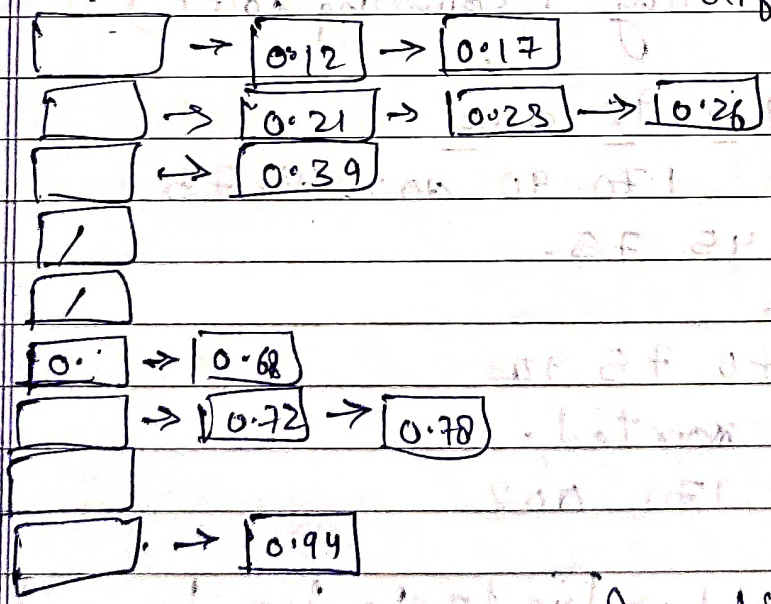
Visualise bucket Sort



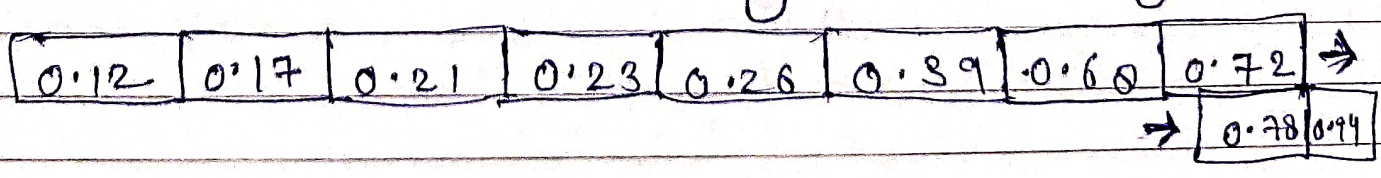
Bucket Array

Sorting Individual Bucket

Each bucket is treated a different Linked List



Inserting into Ascending Order



Page No. ____

Date __/__/____

Saathi

Radix Sort → It is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

Rather than comparing elements directly, Radix sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits from the least significant to the most significant.

Ex → [170, 45, 75, 90, 802].

802 → has three digits so we will iterate three times.

Sort the elements based on the unit place. We use stable sorting (counting sort).

(1) 170 45 75 90 802

Unit digit sorted → 170 90 802 45 75

(2) 170 90 802 45 75.

ten's digit sorted.

802 45 170 75 90

(3) hundred place sorted.

45 75 90 170 802

Sorted → | 45 | 75 | 90 | 170 | 802 |

Page No.

Date __/__/____

Saathi

Counting Sort → It is a non-comparison based sorting algorithm that works well when there is limited range of input values. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the element in their correct sorted positions.

Example →

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

max = 5

Step → 2 CountArray → size = max + 1

0	1	2	3	4	5
0	0	0	0	0	0

Step → 3

0	1	2	3	4	5	
2	0	2	3	0	1	→ frequency

Step → 4 Store the cumulative sum of prefix sum

countArray[i] = CountArray[i-1] + CountArray[i]

0	1	2	3	4	5
2	2	4	7	7	8

Step 5 update outputArray[countArray[inputArray[i]]-1] = inputArray[i].

inputArray =

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

→ 3

Count Array =

0	1	2	3	4	5
2	2	4	7	7	8

→ 7-1=6

Output Array =

0	1	2	3	4	5	6	7
						3	

and so on.....

Page No.

Date __/__/____

Saathi

Step 6 :→ | 0 | 0 | 2 | 2 | 3 | 3 | 3 | 5 |

Check Vs code

Time Complexity = O(N+M) N & M are size of input Array & countA

Best
Worst
Average
→ O(N+M)

Space Complexity → O(N+M)

Advantage →

1) Counting sort generally perform faster than all comparison based.
2) Counting sort is easy to code.
3) Counting sort is stable algorithm.

Disadvantage →

1) Counting sort doesn't work on decimal values.
2) Inefficient if the range of values is very large.
3) It uses extra space.

Remaining Sorting is in Vs Code not so important.